

ITS assignment 1

xdr622 and rsk975

September 2022

SEED

Task 2

In this task we had to encrypt some textfiles using different encryption algorithms. We ran the `openssl enc` command with three different ciphertext algorithms: `aes-128-cbc`, `bf-cbc` and `aes-128-cfb`. We used the following command in the terminal only replacing the `-ciphertext`: `openssl enc -ciphertext -e -in words.txt -out cipher.bin -K 00112233445566778899aabbccdd -iv 0102030405060708` This task helped us get familiarised with encryption in the linux terminal.

Task 3

In this task we encrypted some BMP files using ECB and CBC encryption; the goal was to visually represent the difference between the two.

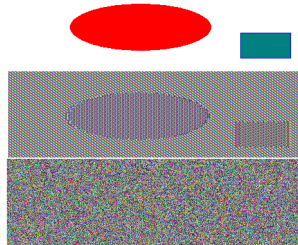


Figure 1: the three pictures of a square and a circle

When encrypted with **ECB**, the original colours of the image are scrambled; however, the contours of the shapes from the original image are still visible. A user may not be able to determine the original colours at a glance, but they will be able to clearly distinguish the existence of a rectangle and an oval in the same positions they appeared in the original picture.

When encrypted with **CBC**, the picture is far more scrambled to more truly resemble random noise. Defining characteristics of the original image appear practically non-existent.

The same experiment with another self-picked picture again it is possible to see that the **ECB** encryption creates a pattern while the **CBC** is just noise. We used the following command for the encrypted of the BMP files, ECB: `openssl enc -aes-128-ecb -e -in nuts.bmp -out nutsecb.bin -K 00112233445566778899aabbccdd` CBC: `openssl enc -aes-128-cbc -e -in nuts.bmp -out nutscbc.bin -K 00112233445566778899aabbccdd -iv 0102030405060708`



Figure 2: The self-picked picture of nuts

Task 4

Modes

ECB and CBC require padding because in these modes the input needs to be a multiple of a blocksize. This is because they use block ciphers, which need to be equally divided. CFB and OFB does not need padding, because they use stream ciphers and do not require dividing the plaintext into blocks to create the ciphertext.

Bytes and padding with CBC

The file containing 5 bytes had a file size when encrypted of 16 bytes and when decrypted it had 11 spaces of padding. The file containing 10 bytes had a file size when encrypted of 16 bytes and when decrypted it had 5 spaces of padding. The last file containing 16 bytes had a file size of 32 bytes when encrypted but did not contain padding when decrypted.

According to the above observations, it appears that `aes-128-cbc` will pad to fill 16 bytes per cipher block.

This exercise was achieved by encrypting the files using the command explained earlier in the report and then decryption those files using `openssl enc -aes-128-cbc -d -in 5bytes.bin -out 5bytes.txt -K 00112233445566778899aabbccdd -iv 0102030405060708 -nopad`. The bytes were checked using a text editor that was able to show the padding and double checked by using hexdump in the terminal.

Task 5

Predictions

ECB should be recoverable everywhere but the first block with the corrupted byte. Because ECB does not rely on the previous decrypted block for decryption,

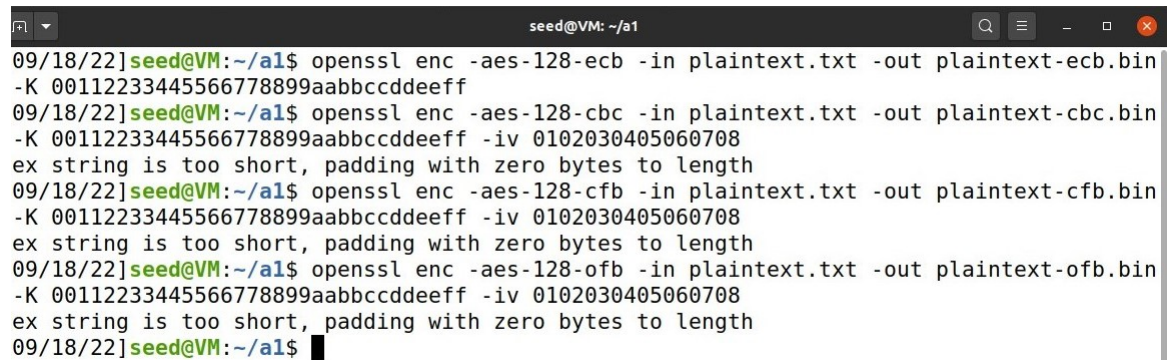
it takes one separate block and decrypts it.

CBC will likely result in a corrupted plaintext output in the block after the one containing the corrupted data, as ciphertext is used to flip the decrypted bits of the next block into plaintext. Afterwards, the corrupted data is no longer used, so the remaining data should be fully recoverable.

CFB uses the entire output of the block cipher; therefore, we predict that the text is recoverable only until the point in which the corrupted data occurs, whereafter all following data will be translated into garbage plaintext output, as the decryption algorithm will continue to use corrupted ciphertext as the IV to the encryption.

OFB should be mostly recoverable since flipping a bit in the ciphertext produces a flipped bit in the output plaintext. In this way, it should resemble ECB.

Encryption and Decryption



```
seed@VM: ~/a1
09/18/22]seed@VM:~/a1$ openssl enc -aes-128-ecb -in plaintext.txt -out plaintext-ecb.bin
-K 00112233445566778899aabbccddeeff
09/18/22]seed@VM:~/a1$ openssl enc -aes-128-cbc -in plaintext.txt -out plaintext-cbc.bin
-K 00112233445566778899aabbccddeeff -iv 0102030405060708
ex string is too short, padding with zero bytes to length
09/18/22]seed@VM:~/a1$ openssl enc -aes-128-cfb -in plaintext.txt -out plaintext-cfb.bin
-K 00112233445566778899aabbccddeeff -iv 0102030405060708
ex string is too short, padding with zero bytes to length
09/18/22]seed@VM:~/a1$ openssl enc -aes-128-ofb -in plaintext.txt -out plaintext-ofb.bin
-K 00112233445566778899aabbccddeeff -iv 0102030405060708
ex string is too short, padding with zero bytes to length
09/18/22]seed@VM:~/a1$
```

Figure 3: Creating the encrypted files

ECB The plaintext output P_2 of the 3rd 16-bit cipher block C_2 has been entirely corrupted. This is due to C_2 being used as input for its own decryption process, which results in significantly different (thus garbage) output. The rest of the text remains intact after decryption, excepting non-ASCII characters.

CBC The plaintext output P_2 of the 3rd 16-bit cipher block C_2 has been entirely corrupted. Similarly to ECB, this is due to its corresponding cipher block C_2 being used as input for its own decryption process, which results in garbage output when XOR'd with the initialisation vector C_1 . In addition, the following 16-bit block C_3 remains almost intact after, save for a random ! in "... want!to discover ..." which is the result of the flipped bit in the prior C_2 'nd ciphertext being XOR'd with otherwise intact ciphertext; the ASCII codes for space and ! are 0x20 and 0x21 respectively, so this follows expectations. The rest of the text is left intact after decryption.

CFB The output of "side" in P_2 is changed to "ride" due to the bit flip. As "r" and "s" are adjacent alphabetically, this follows expectations. The text in the



Figure 4: The files after a flipped bit, left upper corner: ECB. right upper corner: CBC. left lower corner: CFB. right lower corner: OFB

4th plaintext block P_3 is corrupted, as it relies on the ciphertext of the previous block C_2 as the initialization vector for its decryption, and that ciphertext has had a bit flipped; the result of the decryption process is thus significantly different, leading to garbage when XOR'd with P_3 's corresponding cipher block C_3 . The rest of the data is decrypted normally, as the ciphertext blocks C_{i-1} for their corresponding decryption operations $C_i \rightarrow P_i$ have been left intact.

OFB We can see that apart from the word "side" being changed to "ride" in P_2 , the decryption has succeeded. This makes sense as the "r" character is the 55th byte, and since OFB uses the ciphertext as a stream cipher, the flipped bit in the cipher text C_2 corresponds directly to the flipped bit in the plaintext at the same location P_2 . The rest of the plaintext remains intact after decryption.

Short answers

Question 1

Briefly explain Kerckhoffs' principle.

“A system’s security should not rely on the secrecy of its design details.” A system must be secure even if everything is known about it (except the key). Thus, we must always assume the enemy knows how your system works; security should only require protection of your keys, not on knowledge of the algorithm itself.

Question 2

RSA with a key size of 2048 is superior to SHA256, why, why not?

SHA-256 is a secure hashing algorithm, it is used for one-way encryption, checking message integrity and sender authentication. RSA-2048 is an asymmetric cryptography, which encrypts and decrypts, normally in the form a private and public key. Normally things needed to be read by humans are encrypted using RSA while things only read by computers are encrypted by SHA. Thus these two are not used in the same cases, and can not be compared like that. The key-size would then not matter.

Question 3

Do we have to use a key with a fixed size in HMAC? If so, what is the key size?

If not, why? Not a fixed size, because it needs to match the output and that can differ. The key just needs to be a blocksize but not a specific number; precisely, the key must be both a multiple of a byte, and equal to the output size.

Question 4

AES in CBC mode of operation is better than AES in ECB. Why?

Because ECB is given a key k which is used to encrypt several identical plaintext blocks, this creates patterns in the encryption. CBC introduces a changing IV (initialization vector) that disrupts these patterns making the ciphertext more random; it uses the ciphertext of one block C_{i-1} as the initialisation vector for the next block C_i to be encrypted. The attacker does not just have to know the IV, but also the right order in which it is used.